Atlan backend challenge (Datastore)

Overview:

We need to build a system that allows the user to store multiple types of forms and responses. The user can integrate 3rd party platform with the form response. We should design a system such that we can onboard any 3rd party integration in a “plugged-in” fashion. It should be less coding and more configuration work for any new integration.

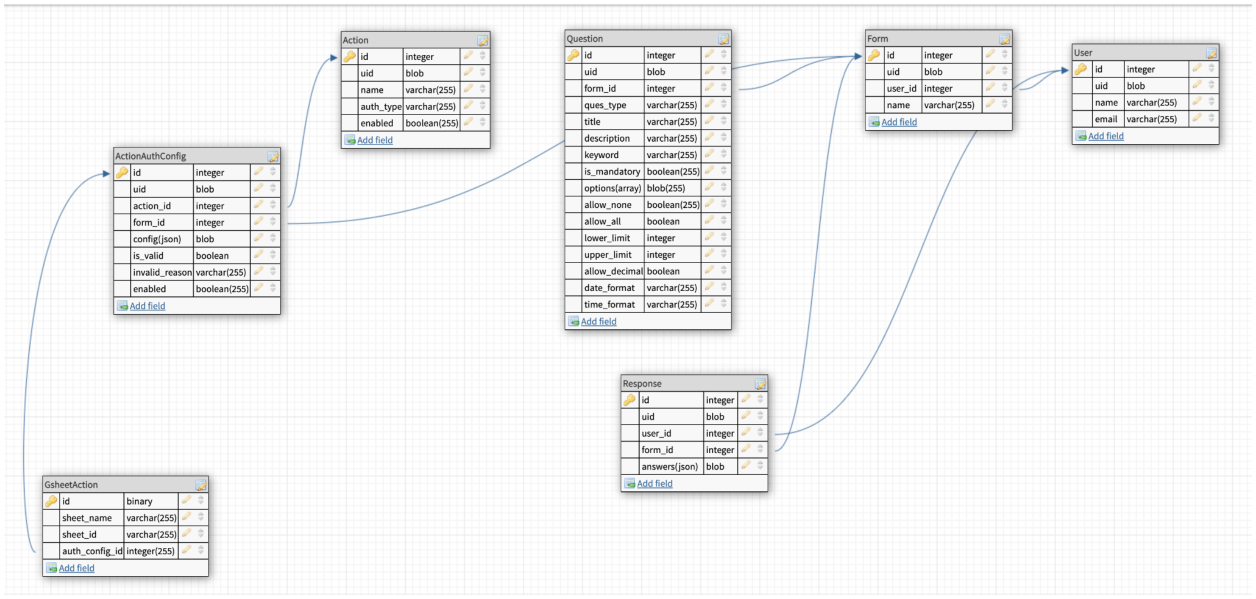
Design idea:

The idea is to build a generic platform with the help of a reusable component. The reusable component should be used by any new integration. We need to build this system gradually. Whenever we onboard new integration, we should add only core logic specific to integration. We should not expect to build a complete generic system with the first integration as we are not aware of all the use cases. This platform becomes robust and reliable as we onboard new integration.

Tech stack used to build the system:

- Python
- Django framework
- Postgres
- Django rest framework
- Celery
- Redis (Celery broker)

Database design(schema):



Action

id	integer
uid	blob
name	varchar(255)
auth_type	varchar(255)
enabled	boolean(255)

Question

id	integer
uid	blob
form_id	integer
ques_type	varchar(255)
title	varchar(255)
description	varchar(255)
keyword	varchar(255)
is_mandatory	boolean(255)
options(array)	blob(255)
allow_none	boolean(255)
allow_all	boolean
lower_limit	integer
upper_limit	integer
allow_decimal	boolean
date_format	varchar(255)
time_format	varchar(255)

Form

id	integer
uid	blob
user_id	integer
name	varchar(255)

User

id	integer
uid	blob
name	varchar(255)
email	varchar(255)

ActionAuthConfig

id	integer
uid	blob
action_id	integer
form_id	integer
config(json)	blob
is_valid	boolean
invalid_reason	varchar(255)
enabled	boolean(255)

Response

id	integer
uid	blob
user_id	integer
form_id	integer
answers(json)	blob

GsheetAction

id	binary
sheet_name	varchar(255)
sheet_id	varchar(255)
auth_config_id	integer(255)

Note: Please visit this public URL to check the DB design

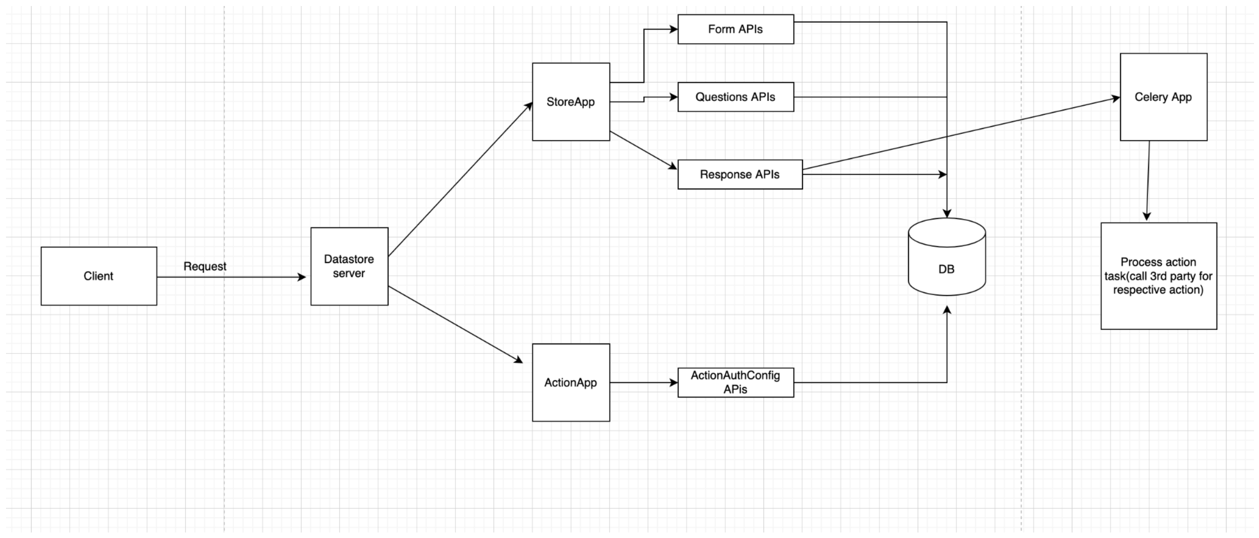
Platform and codebase design:

Project name: datastore

App names: store, action

Store app: The primarily managing the form & response data. Whenever there is any new response added to DB, it triggers the respective action in the action app.

Action app: The primarily managing action and their authentication. It runs action that is attached to any form responses.



Google sheet integration & code walkthrough:

We will walk through the codebase and try to see how we have integration google-sheet integration. We will cover only import files and aspects of the codebase. The codebase repo can be found here.

datastore/settings.py: Here we are storing project configurations. We need to update the database, celery and Redis configuration in this file.

datastore/celery.py: It has a celery configuration for the async tasks.

store/models.py: It has User, Form, Question, and Response models.

store/tasks.py: This file has an async function *process_form_response,* It takes response_id as input and calls the appropriate function in 3rd party integration module. This function will retry 5 times if the action fails for any reason.

store/serializers.py: Here we have added the serializer that is needed to store APIs.

action/models.py: It has Action, ActionAuthConfig, and GsheetAction models

action/common_utils/oauth2base.py: As google sheet integration uses OAuth2 type, we have generic code for oauth2 grant flow. Any integration that uses OAuth2 should be able to reuse the same code.

action/common_utils/action_config.py: Here we store serializer_map and action_map for each integration.

action/googlesheet/config.py: A google sheet integration-specific configuration is stored here. It mostly has OAuth2-specific details.

action/googlesheet/oauth2.py: A google sheet specif code for OAuth2 flow. It overrides the base OAuth2 code.

action/googlesheet/process_action: Here we declare a call *GoogleSheetAction*, this class has a method *process_action*, this method creates or updates the google sheet data. We call the *process_action* method whenever there is any response to a googlesheet integrated form.

How to onboard a new integration (or Action)?

Usecase: Send an SMS to the customer whose details are collected in the response as soon as the ingestion was complete reliably.

Assuming the action name is sms_trigger

Steps:

- Add a new entry to the *Action* table with the name sms_trigger.
- Add *sms_trigger/config.py* to specify the authentication configuration if needed
- Add authentication mechanism in *sms_trigger/auth.py*
- Register serializer and action classes in *action/common_utils/action_config.py*

Advantages of the above design:

- Maintainability and Logging are so easy as there will be only 1 backend service running.
- As we are having reusable components, the will be no code duplication.
- Need less time to build this platform with this design.

Disadvantages:

- The codebase size becomes too big and unreadable as we continue to onboard new integration.

- As the questions and responses vary from form to form, the use of the relational database Postgres can decrease the performance when we have a huge data in DB.

Other design ideas:

- We can design a system such that each integration will have its own set of rules. We can declare these rules in YMAL format. The rules will contain the definition of authentication, process_action etc.

  Pros:

  - By adding just one YAML script will enable the new integration to the platform.

  - Maintaining the service would be easy.

  Cons:

  - It will be hard to build a parser that can understand YAML script rules and take the appropriate action.

  - The YAML rules may not be readable to every developer. It would be hard for new team members to understand the platform.

- We can design our platform in microservice. Each new integration will be a new microservice. We will have one service that will connect with the rest of the microservice. This service will act as an interface for the platform. All requests and responses will pass through this microservice.

  Pros:

  - The system will be more secure as all requests pass through a single interface. The core services running behind the core services.

  - As the codebase split into different microservices, It would be more readable as compared to other design ideas.

Cons:

- Maintaining would be difficult as there will be multiple services running.

- The cost of the platform will increase as there are multiple services using multiple resources.